Undoing Event-Driven Adaptation of Business Processes

Sébastien Mosser, Gabriel Hermosillo, Anne-Françoise Le Meur, Lionel Seinturier, Laurence Duchien
INRIA Lille Nord Europe — University of Lille 1 — Laboratoire LIFL - CNRS UMR 8022
Lille, France — Email: {Firstname.Lastname}@inria.fr

***Abstract*—As business processes continue to gain relevance in different domains, dynamicity is becoming a great concern. Static processes no longer cover the actual needs of constantly changing environments, and process adaptation is a must in order to maintain competitive levels. While creating dynamically adaptable business processes can be a challenging task, undoing these adaptations is a natural functionality that has not been studied in depth. Straight forward approaches for unadaptation can easily end up with corrupted processes, bringing uncertainty to the whole business logic. In this paper we bring forward a solution for efficiently undoing a business process adaptation in event-driven environments, considering also the correlated adaptations that happened afterwards.**

***Keywords*-Event Driven Approach; Business Process; Adaptation;**

I. Introduction

There is a huge amount of variable conditions surrounding today's business environments. The context in which our business processes (Bps) are executed is an important factor that directly affects the way they are executed. We can no longer rely on static processes, since we need to consider these conditions and adapt our processes accordingly in order to obtain better results and maintain a competitive level. Several efforts have been made towards adding dynamicity to Bps, showing how much this problematic matters [1]–[4].

By monitoring the context in which business processes are being executed, it is possible to efficiently respond to any changes in the environment and continue the process in an optimal way. Changes in the context can be seen as events that arrive at a specific moment in time and have a different meaning depending on several conditions, *e.g.*, timing, origin, sequence. The meaning of each event can help us determine a specific situation to which we can react by adapting the process. However, sometimes it is not a single event, but a combination of events that should trigger the adaptation, and this is where *Complex Event Processing* (Cep) can be used. Cep is an emerging technology from which organizations can benefit, since it allows them to find real-time relationships between different events, using elements such as timing, causality, and membership in a stream of data to extract relevant information [5].

Using Cep approaches, the reception of an event ϵ (*e.g.*, a CPU overload) meet a certain condition and the system is then adapted [6]. But, what happens when we receive $\neg\epsilon$? The condition is no longer true (*i.e.*, the CPU load is back to normality) and we would like to go back to our original process. Undoing of adaptations is a topic that has been left aside, but it cannot be obviated, since, as we will explain in this paper, a straight forward approach can easily lead to corrupted processes.

The objective of this paper is to present an effective solution to event–driven Bp unadaptation, by considering not only the event that caused the adaptation, but also the correlated adaptations that came afterwards, leaving all the unrelated adaptations untouched, in order to obtain a Bp *"as it would be if this adaptation had never happened"* (similarly to transactional systems [7] where the *rollback* operation is used to restore a system). Using this generic and automated approach, users are relieved from handling the unadaptation logic.

The rest of this paper is organized as follows. In Section II, we use a scenario to illustrate the motivation and challenges of our proposal. Section III presents our approach for doing and undoing Bp adaptations. Section IV describes how the actual undoing of Bp adaptation is achieved. In Section V, we show an implementation to validate our proposal. Section VI gives some related work. Finally, Section VII concludes and discusses future work.

II. Motivation and Challenges

In this section we will use a simple example to illustrate how a business process can be adapted and why undoing this adaptation is needed, but is not a simple and transparent task.

A. Example Description

We consider here a simple process, part of an online catalog software. It contains five activities, which respectively: *(i)* logs the user in, *(ii)* asks for user's request, *(iii)* performs the search in the internal database, *(iv)* displays the results to the user and finally *(v)* logs the user out. This process is depicted in Fig. 1.

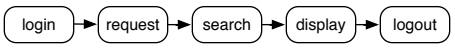


Figure 1. Illustrative business process (initial)

We want to adapt this process according to the context, using an event-driven approach. Process adaptations are driven by the reception of explicit events (triggered by associated conditions). For example, if the search service becomes unavailable, a *fail* event will be triggered, and an

adaptation will be executed to fix the problem. Precisely, it will connect the process to a remote backup service, to ensure continuity for customers. We summarize in TAB. I the different adaptation rules associated to our running example.

Event	Condition	Action
fail	$search\_status \neq ok$	Use a backup server
slow	$bw < 100kbps$	—
cache	**fail** *followed by* **slow**	Introduce a cache
perf	$cpu > 80\%$	Monitor the process

Table I
EVENT–DRIVEN ADAPTATION DECISIONS

Accordingly, if the *fail* event is received, the business process will be adapted to tackle this issue, and we will obtain after the adaptation the process depicted in FIG. 2. In this figure (and the upcoming ones), we represent deleted elements with *dashed* lines .

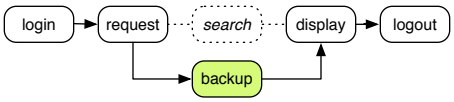


Figure 2. Consulting a backup when the search service is unavailable

If we then receive a performance alert by the event *perf* (identifying a CPU abnormal usage), we want to start monitoring the CPU consumption for all the activities in the process. To achieve this, we will add a monitoring activity after each existing activity. The resulting process is depicted in FIG. 3.

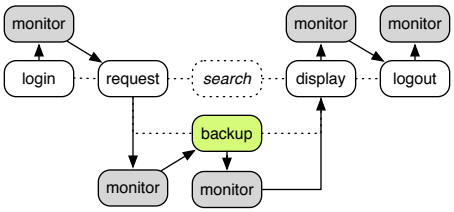


Figure 3. Monitoring the process to identify abnormal CPU consumption

As the backup server is a remote entity, we depend on the quality of the network connection to search the catalog. Considering a bandwidth drop (identified with a *slow* event), the *cache* event will also be recognized (as it is defined as a *fail* followed by a *slow*) and we will need to adapt the process by adding a cache mechanism to help diminish the response time of the requests. The adapted process is depicted in FIG. 4.

B. Need for Adaptation Undo

When an adaptation condition is no longer true, we would like to get our process *"as it would be if this adaptation had never happened"*. Retaking our previous example, let's say we receive an event $\neg$*fail*[1], which means that we recovered our internal search server. In this case, we no longer need the external backup nor the associated cache mechanism and we can remove them.

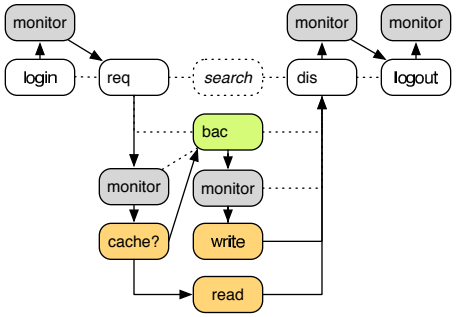


Figure 4. Introducing a cache to deal with lower bandwidth

Naively, undoing an adaptation does not seem so complicated. It can be seen as removing all the changes made to the business process that were caused by the *fail* event. In order to achieve this, the intuitive undoing action would be to use the exact "opposite" of the used adaptation. In our case, it would remove the backup server and re–introduce the internal search one. The associated process is depicted in FIG. 5(a). Unfortunately this process does not make sense in terms of business logic, as it holds the two following issues[2]: *(i)* the *search* activity is not monitored and *(ii)* the cache mechanisms are irrelevant since the vanishing of the backup server.

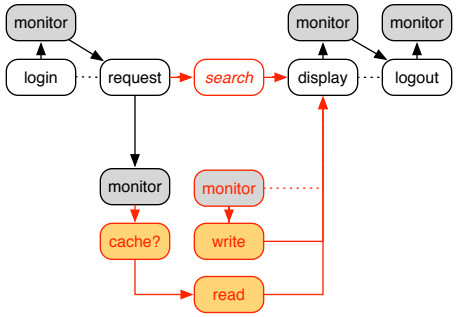
C. Challenge: Automating Adaptation Undoing

As seen in the previous example, a straight forward undoing of the adaptation can result in a corrupted process. To obtain a correct undoing of an adaptation, we could add an adaptation rule that changes the process to its original state. However, this approach will only work if we consider all the possible states of the process, given all the possible adaptations that could happen, providing the correct process for each and every one of them. This, far from being user friendly, is virtually impossible to accomplish.

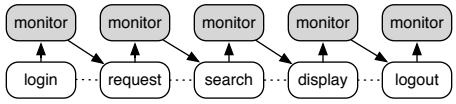
The ideal case would be to provide the user with an automated unadaptation of the process, whenever adaptation conditions are no longer met. Using this approach, we could automatically produce a system *"as it would be if this adaptation had never happened"*. Going back to the previous example, it results in the synthesis of the process depicted in FIG. 5(b): the search activity is monitored, and the cache

[1]Intuitively, if *fail* is defined as $search\_status \neq ok$, $\neg$*fail* is defined as $search\_status = ok$

[2]Syntactically talking, the removal of the backup activity also creates a hole between the cache validity test and the cache writing activity, leading to a corrupted process.



(a) Inconsistent process obtained after naive unadaptation



(b) Expected result

Figure 5. Undoing adaptation ($\neg$*fail*): a *not–so–easy* task

mechanism is not present (since its trigger event depends on the *fail* one). As a consequence, to support this automated unadaptation, we highlight the following mechanisms M_i, which are required to properly undo an adaptation triggered by an event ϵ:

- M_1: **Identify the undoing trigger**. Based on the description of events, the system must be able to recognize their opposite, and trigger the automated undoing mechanisms when relevant.
- M_2: **Restore the process**. The current process must be restored to what it was *before* the reception of ϵ.
- M_3: **Forget the correlated adaptations**. Adaptations triggered by any event which depends (immediately or transitively) on ϵ (*e.g.*, the **`cache`** event depends on the **`fail`** event) must be forgotten.
- M_4: **Re–execute the unrelated adaptations**. All adaptations that are independent of ϵ must be re–executed, to yield a system equivalent to the one obtained after their on–the–fly execution.

III. Adaptation: From "*do*" to "*undo*"

To tackle the issues identified in the previous section, we propose to automate the support of BP unadaptation. The key idea is to formalize the adaptation, and to rely on this formal model to define and then operationalize the unadaptation. We consider here an event–driven adaptation engine based on state–of–the–art mechanisms [8], represented in Fig. 6. At a coarse–grained level, the engine receives a continous flow of events from deployed sensors. According to the received events, the CEP engine will trigger associated adaptations, stored in an adaptation repository. The obtained (*adapted*) process is then sequentially used as input for the upcoming adaptations.

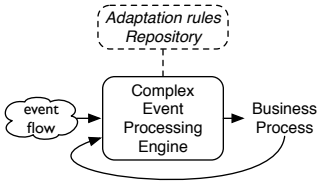
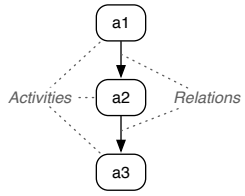


Figure 6. Overview of the adaptation process

A. Business Processes & Actions

We define a business process $p \in \mathcal{P}$ as a set of activities $acts$, which implement elementary tasks, and a set of causal relations $rels$ to schedule the activity set according to a partial order. This formalization is based on the ADORE meta–model [9] and models a subset of the BPEL industrial standard [10]. For simplification purpose, we assimilate an activity to its $name$, without further knowledge of its internal contents. A (binary) causal relation is defined as an ordered pair of activities (*i.e.*, $left$ and $right$). We denote as $left \prec right \in \mathcal{R}$ the fact that a relation exists between $left$ and $right$. We depict in Fig. 7 a business process and its associated formal representation.



$$p = (\underbrace{\{a_1, a_2, a_3\}}_{acts}, \underbrace{\{a_1 \prec a_2, a_2 \prec a_3\}}_{rels})$$

Figure 7. A simple business process, $p \in \mathcal{P}$.

To manipulate BPs, we use an action–based approach, since these approaches are known to efficiently support the manipulation of models [11]. An elementary action is defined as the addition or deletion of a model element in a given business process. In itself, an action α is simply a ground term that reifies the associated intention (*e.g.*, adding an activity, deleting a relation). Tab. II lists the existing actions available to modify a given business process. The execution of an action α on a process p is handled by a call to the $exec$ function: $exec(\alpha, p) = p'$, where p' is a process effectively modified.

Intention	Notation
Add an activity a	$add_a(a)$
Add a relation $a \prec a'$	$add_r(a, a')$
Del an activity a	$del_a(a)$
Del a relation $a \prec a'$	$del_r(a, a')$

Table II
ACTIONS AVAILABLE TO MANIPULATE BP

Actions can be sequenced to implement complex modifications. Let $A = [\alpha_1, \ldots, \alpha_n]$ be a sequence[3] of actions. The execution of A on a given process p is formally defined as follows:

$$exec^+(L,p) = \begin{cases} L = \emptyset & \Rightarrow \quad p \\ L = [\alpha|A] & \Rightarrow \quad exec^+(A, exec(\alpha, p)) \end{cases}$$

Using this representation, the process p depicted in FIG. 7 can be built as the result of the execution of its associated action sequence A_p on the empty process.

$$\begin{aligned} A_p &= [add_a(a_1), add_a(a_2), add_r(a_1,a_2), \\ &\qquad add_a(a_3), add_r(a_2,a_3)] \\ p &= exec^+(A_p, (\emptyset, \emptyset)) \end{aligned}$$

B. Events & Complex Events

An event ϵ is reified as a value associated to a sensor, *e.g.,* the *CPU load* is equal to *80%*. We use a tuple of terms $\epsilon = (sensor, value)$ to represent this information. Events are continuously sent to a complex event processor engine, in our case Esper[4]. Based on this continuous flow, this engine identifies *complex events* (CES), defined as *(i)* a boolean formula applied to an (elementary) event to process it or *(ii)* a combination of other complex events. We represent in TAB. III the expressiveness associated to usual complex events definitions. CES can be conjucted ($\wedge$) or disjuncted ($\vee$) using elementary boolean logic. A sequence operator is used to introduce causality between two events ($\epsilon_1;\epsilon_2$ means that ϵ_2 foolows ϵ_1 eventually, even if not immediately). Finally, a time window operator supports the wait for a given complex event for a given duration (*e.g.*, $\epsilon' = within(\epsilon, 200ms)$ will be recognized if the CE ϵ is received by the engine within $200ms$).

Intention	Notation	Example
Event processing	$(sensor \sim value)$	$(cpu{>}80\%)$
CE conjunction	$\epsilon_1 \wedge \epsilon_2$	$slow \wedge (error{=}404)$
CE disjunction	$\epsilon_1 \vee \epsilon_2$	$(error{=}404) \vee (error{=}503)$
CE sequence	$\epsilon_1;\epsilon_2$	$fail;slow$
Time window	$within(\epsilon, \Delta_t)$	$within(\neg response, 10s)$

Table III
COMPLEX EVENTS DEFINITION

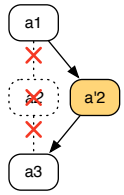
C. Event–driven BP Adaptation

Based on these definitions of BPs and CEs, one can add adaptability into existing business processes. To perform such a task, a user would define adaptation rules, and store them into a shared rule repository. This repository is connected to the complex event processing engine, which triggers the adaptation application at runtime.

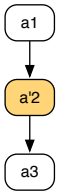
An adaptation rule $r \in \mathcal{A}_\mathcal{R}$ is defined as a tuple (ϵ, φ), where ϵ is the CE used to trigger the adaptation[5], and φ is a function used to compute the action sequence to be executed to perform the adaptation. This action sequence is executed on the BP, to modify its structure and then implement the adaptation:

$$\text{Let } a = (\epsilon, \varphi) \in \mathcal{A}, p \in \mathcal{P}, \ \epsilon \Rightarrow exec^+(\varphi(p), p)$$

We illustrate such an adaptation in FIG. 8. The goal of this adaptation is to replace an activity by another one when the ϵ complex event is processed (*e.g.,* the replacement of the internal search by the backup server in the running example, FIG. 2). The application of φ on the process p produces a sequence of actions A, which aims to replace the activity a_2 by a new activity a'_2. To implement this adaptation, the engine executes A on p, and computes as output p', the adapted process.



(a) $\varphi(p) = A$



(b) $p' = exec^+(A, p)$

$$\begin{aligned} A = \ & [add_a(a'_2), add_r(a_1,a'_2), add_r(a'_2,a_3), \\ & del_r(a_1,a_2), del_r(a_2,a_3), del_a(a_2)] \end{aligned}$$

Figure 8. Applying an adaptation (ϵ, φ) to p

To properly support unadaptation of BPs, we need to keep track of the adaptation history. This concept is expressed as a list of tuples (ϵ, A_ϵ), where ϵ is the processed CE and $A_\epsilon = [\alpha_1, \ldots, \alpha_n]$ the sequence of actions computed according to this event. The list is maintained in reversed order, *i.e.*, the head of the history corresponds to the last adaptation. Considering the final adapted process (FIG. 4) associated to the adaptation scenario depicted in SEC. II, the history is defined as follows:

$$H_{ex} = [(perf, A_{perf}), (slow, A_{slow}), (fail, A_{fail})]$$

D. Automating Adaptation Undoing

We consider here the situation depicted in FIG. 9. This situation is a formal representation of the adaptation scenario textually described in SEC. II: p is the initial scenario

[3]We assimilate a sequence to a totally ordered set (*i.e.,* a list), and use the notation and functions associated to lists usually encountered in the logic programming literature [12]. A list is defined as a head h followed by a tail list T, and is denoted as $l = [h|T]$. The empty list is $\emptyset$.

[4]http://esper.codehaus.org/

[5]We assume that a given complex event ϵ will trigger only one adaptation: $\forall(\epsilon,\varphi) \in \mathcal{A}, \nexists(\epsilon',\varphi'), \ \epsilon = \epsilon'$. According to state–of–the–art engines, we also assume that an adaptation is only triggered once.

(FIG. 1), and p_{123} is the resulting adapted process (FIG. 4).

$$p \overset{\epsilon_1}{\longmapsto} p_1 \overset{\epsilon_2}{\longmapsto} p_{12} \overset{\epsilon_3}{\longmapsto} p_{123}$$

Defined complex events: $\{\epsilon_1, \epsilon_2, \epsilon_3\}$
Complex events combination: $\epsilon_3 = \epsilon_1; \epsilon_2$
Rule repository: $\{(\epsilon_1, \varphi_{\epsilon_1}), (\epsilon_2, \varphi_{\epsilon_2}), (\epsilon_3, \varphi_{\epsilon_3})\}$
Adaptation steps:
- $p_1 = exec^+(A_{\epsilon_1}, p),\ A_{\epsilon_1} = \varphi_{\epsilon_1}(p)$
- $p_{12} = exec^+(A_{\epsilon_2}, p_1),\ A_{\epsilon_2} = \varphi_{\epsilon_2}(p_1)$
- $p_{123} = exec^+(A_{\epsilon_3}, p_{12}),\ A_{\epsilon_3} = \varphi_{\epsilon_3}(p_{12})$

History: $[(\epsilon_3, A_{\epsilon_3}), (\epsilon_2, A_{\epsilon_2}), (\epsilon_1, A_{\epsilon_1})]$

Figure 9. Doing adaptation: p becomes p_{123}.

Doing adaptation. The adaptation rule repository holds three rules, defined with respect to three complex events: $\{\epsilon_1, \epsilon_2, \epsilon_3\}$. Complex events ϵ_1 and ϵ_2 are elementary event processing, and ϵ_3 is defined as the detection of ϵ_2 *after* the detection of ϵ_1. Based on the analysis of the incoming elementary events, adaptations are triggered by the complex event processing engine to adapt a given process p. We consider here the following sequence of events: ϵ_1, ϵ_2, and consequently ϵ_3 (according to its definition). After these three adaptations, we obtain a process p_{123}. This process is handled through the previously defined mechanisms.

Undoing adaptation. We consider now the detection of a complex event opposed to ϵ_1 (denoted as $\neg\epsilon_1$). In this new context, adaptations that had been triggered based on ϵ_1 do not make sense anymore, and must be undone. Considering that our objective is *"to produce the system as it would be if this adaptation had never happened"*, we also need to undo all adaptations depending on ϵ_1 (*i.e.*, triggered by a complex event which combines ϵ_1 with others, here ϵ_3). According to this goal, and with respect to the mechanisms M_i identified in SEC. II, the system needs to *(i) recognize* the opposite event and then trigger the undoing mechanisms (M_1), *(ii) rewind* the historic to retrieve the process as it was before the reception of the incriminated event (M_2), *(iii) prune* from the historic the adaptations that depend (immediately or transitively) on this event (M_3), and finally *(iv) replay* the remaining adaptations to obtain the expected process (M_4).

Considering the example depicted in FIG. 9, the undo mechanism associated to ϵ_1 is expected to automate the following steps:

- *recognize*: Assuming that ϵ_1 is an arithmetic comparison (*e.g.*, $bandwidth < 100kbps$), its opposite can be automatically computed (*i.e.*, $\neg\epsilon_1 = bandwidth \geq 100kbps$). An adaptation needs to be undone since the complex event processor engine detects the event and its opposite in sequence, *i.e.*, $\epsilon_1^{-1} = \epsilon_1; \neg\epsilon_1$.
- *rewind*: On the detection of ϵ_1^{-1}, the system will restore the process as it was before the detection of ϵ_1. In our case, this *rewind* restores the process as p.
- *prune*: Considering the contents of the history, the engine knows that the process p was adapted according to the following sequence of events: $[\epsilon_1, \epsilon_2, \epsilon_3]$. The pruning step removes from this sequence the incriminated event, and all its (transitive) dependencies (here, ϵ_3). In our case, the pruned sequence is $[\epsilon_2]$.
- *replay*: The adaptations triggered by the events contained in the pruned sequence need to be replayed on the process. In our case, it means to adapt p according to the rule associated to ϵ_2.

We obtain as a result of this adaptation undoing the process p_2, as represented in FIG. 10.

$$p \underbrace{\overset{\epsilon_1}{\longmapsto} p_1 \overset{\epsilon_2}{\longmapsto} p_{12} \overset{\epsilon_3}{\longmapsto}}_{do}\ p_{123}\ \underbrace{\overset{\neg\epsilon_1}{\longmapsto} (p \overset{\epsilon_2}{\longmapsto})}_{undo}\ p_2$$

$$p_2 = exec^+(\varphi_{\epsilon_2}(p), p)$$

Figure 10. Undoing adaptation: p_{123} becomes p_2

IV. "UNDO" OPERATIONALIZATION

In this section, we formally describe how the undaptation can be operationalized. We present the different operations used to support the $undo$ process using a functional style, being consequently language independent.

A. M_1: Recognition of an Undo Trigger (ϵ^{-1})

We denote as ϵ^{-1} the CE that triggers an undo. This event is defined as the sequence composed by the event ϵ and its associated opposite event $\neg\epsilon$. Using this definition, an undo will always be triggered when the engine recognizes an opposite event (*e.g.*, $\neg fail$) eventually preceded by an event (*e.g.*, *fail*).

$$fail^{-1} = fail; \neg fail$$

We represent in TAB. IV the way an oposite event $\neg\epsilon$ is computed with respect to an event ϵ.

Complex Event (ϵ)	Opposite Event ($\neg\epsilon$)
$(sensor \sim value)$	$(sensor \not\sim value)$
$\epsilon_1 \wedge \epsilon_2$	$\neg\epsilon_1 \vee \neg\epsilon_2$
$\epsilon_1 \vee \epsilon_2$	$\neg\epsilon_1 \wedge \neg\epsilon_2$
$\epsilon_1; \epsilon_2$	$\neg\epsilon_1 \wedge \neg\epsilon_2$
$within(n, \epsilon_1)$	$\neg\epsilon_1$

Table IV
COMPLEX EVENTS (ϵ) & OPPOSITES ($\neg\epsilon$)

B. M_2: Rewinding a Business Process

The objective of this function is to restore a process as it was before the reception of the initial event ϵ. It intensively relies on the history model previously defined, identifying the actions to be undone *and* the encountered events.

Undoing actions. For each kind of action α, we present in TAB. II its inverse α^{-1}. Executing α^{-1} after α annihilates the introduced modification: $exec(\alpha^{-1}, exec(\alpha, p)) = p$. Considering a sequence of actions A, its inverse (denoted as A^{-1}) is defined as the inverse of all actions contained by A, in reversed order. This approach is inspired by aspect unweaving techniques [13].

α	α^{-1}
$add_a(a)$	$del_a(a)$
$add_r(a, a')$	$del_r(a, a')$
$del_a(a)$	$add_a(a)$
$del_r(a, a')$	$add_r(a, a')$

$$\begin{array}{lcl} A & = & [\alpha_1, \ldots, \alpha_n] \\ A^{-1} & = & [\alpha_n^{-1}, \ldots, \alpha_1^{-1}] \end{array}$$

Table V
ACTIONS (α) & INVERSE (α^{-1})

Function description. This operation is implemented in a $rewind$ function. Based on a given process p, the associated history H and the intended event ϵ, this function computes a process p' (representing the business process p as it was before the reception of ϵ) and a list of complex events $[\epsilon_i, \ldots, \epsilon_j]$ (representing all the events recognized between the receptions of $\neg\epsilon$ and ϵ). For clarity reasons, we decouple the computation of p' (using a $restore$ function) from the identification of the encountered events (using an $extract$ function). The definition of these functions, based on the formalization described in SEC. III is presented in FIG. 11.

$$\begin{array}{rcl} rewind : \mathcal{P} \times History \times \text{CE} & \rightarrow & \mathcal{P} \times [\text{CE}] \\ (p, H, \epsilon) & \mapsto & (p', [\epsilon_i, \ldots, \epsilon_j]) \\ restore : \mathcal{P} \times History \times \text{CE} & \rightarrow & \mathcal{P} \\ (p, H, \epsilon) & \mapsto & p' \\ extract : History \times \text{CE} & \rightarrow & [\text{CE}] \\ (H, \epsilon) & \mapsto & [\epsilon_i, \ldots, \epsilon_j] \end{array}$$

$$restore(p, H, \epsilon) = \begin{cases} H = \emptyset \Rightarrow p \\ H = [(\epsilon, A_\epsilon)|H'] \\ \quad \Rightarrow exec^+(A_\epsilon^{-1}, p) \\ H = [(\epsilon', A_{\epsilon'})|H'], \epsilon' \neq \epsilon \\ \quad \Rightarrow restore(exec^+(A_{\epsilon'}^{-1}, p), H', \epsilon) \end{cases}$$

$$extract(H, \epsilon) = \begin{cases} H = \emptyset \vee H = [(\epsilon, A_\epsilon)|H'] \\ \quad \Rightarrow \emptyset \\ H = [(\epsilon', A_{\epsilon'})|H'], \epsilon' \neq \epsilon \\ \quad \Rightarrow [\epsilon'|extract(H', \epsilon)] \end{cases}$$

$$rewind(p, H, \epsilon) = (\ restore(p, H, \epsilon),\ extract(H, \epsilon)\ )$$

Figure 11. Description of the $rewind$ function

C. M_3: Pruning the Adaptation History

The objective of this operation is to identify in a sequence of events the ones related to the to–be–removed event (immediately or transitively), and consequently reject them all (as they are now irrelevant). We define a $prune$ function to support this operation. Using as inputs the sequence of events computed by $rewind$ (named $History$) and a sequence of events to be rejected (named $Removed$, and initially containing the to–be–removed event), this function produces a $Pruned$ sequence of events. According to our objectives, the $Pruned$ sequence contains events that are not related to the ones defined in $Removed$. Its definition is represented in FIG. 12

$$\begin{array}{rcl} prune : [\text{CE}] \times [\text{CE}] & \rightarrow & [\text{CE}] \\ (Hist, Removed) & \mapsto & Pruned \end{array}$$

$$prune(H, R) = \begin{cases} H = \emptyset \Rightarrow \emptyset \\ H = [\epsilon|H'] \wedge \exists \epsilon' \in R, \epsilon \in \epsilon' \\ \quad \Rightarrow prune(H', [\epsilon|R]) \\ H = [\epsilon|H'] \wedge \nexists \epsilon' \in R, \epsilon \in \epsilon' \\ \quad \Rightarrow [\epsilon|prune(H', R)] \end{cases}$$

Figure 12. Description of the $prune$ function

D. M_4: Replaying a Complex Event Sequence

The objective of this operation is to perform process re–adaptation, *i.e.,* to re–execute on the rewinded process the adaptations that still need to be present in the expected result (*i.e.,* the adaptations triggered by the events identified by the $prune$ function). This operation is described in a function named $replay$, presented in FIG. 13. Using a given process p' and a sequence of events $[\epsilon_i, \ldots, \epsilon_j]$ as inputs, the function produces a process p_r that implements the expected result of the undo process.

$$\begin{array}{rcl} replay : \mathcal{P} \times [\text{CE}] & \rightarrow & \mathcal{P} \\ (p', [\epsilon_i, \ldots, \epsilon_j]) & \mapsto & p_r \end{array}$$

$$replay(p, L) = \begin{cases} L = \emptyset \Rightarrow p \\ L = [\epsilon|L'] \\ \quad \Rightarrow replay(adapt(p, \epsilon), L') \end{cases}$$

Figure 13. Description of the $replay$ function

V. VALIDATION & IMPLEMENTATION

The unadaptation function takes as inputs the business process p, the associated history H and the event to unadapt ϵ. Let (p', E) be the result of $rewind(p, H, \epsilon)$, and $pruned$ the result of $prune(E, [\epsilon])$. The unadapted process p_u is obtained as the result of $replay(p, reverse(pruned))$. We implemented the complete approach using the PROLOG language, to support its application on large examples[6]. Events are sent to the adaptation engine through the **`send_events([[sensor, value],...])`** command. We consider here the scenario described in SEC. II, *i.e.,* the reception of an errorneous status for the **`search`** service, followed by a CPU overload and a network slowdown.

```
?- send_events([[search_status, error], ...]).
```

[6]Video demonstration available here: http://bit.ly/scc11

```
%   Recognizing <EP: fail>
%     > act1 = [add_a(backup), ...]
%     > exec+(act1, p) ... done.
%   Recognizing <EP: perf>
%     > act2 = [add_a(m1),  ...]
%     > exec+(act2, p) ... done.
%   Recognizing <EP: slow>
%   Recognizing <EC: cache>
%     > act3 = [add_a(cache?), ...]
%     > exec+(act3, p) ... done.
% => Consumed logical inferences: 2,668 (0.001ms)
?-
```

The undoing is triggered with the recognition of a $\neg fail$ event, *i.e.*, the reception of a $(search\_status, ok)$ event. It first triggers the rewind of the process to its original state. Then, the pruning function removes the $cache$ event as it depends on *fail*. Finally, the replay is triggered, and the expected result (depicted in FIG. 5(b)) is obtained as output.

```
?- send_events([[search_status, ok]]).
%   Recognizing <EP: not(fail)>
%   Undo required <fail>
%     rewind:
%      > exec+(invert(act3), p),
%      > exec+(invert(act2), p),
%      > exec+(invert(act1), p).
%      events = [cache, perf]
%     prune:    [perf]
%     replay:   [perf]
%      > act4 = [add_a(m6), ...]
%      > exec+(act4, p) ... done.
% => Consumed logical inferences: 1,971 (0.001ms)
?-
```

The immediate advantage of our approach is to relieve the user from handling the unadaptation logic. In this example, the recognition of the $\neg fail$ CE is fully automated, based on the value received through the associated sensor. The proposed unadaptation function properly handles this situation, and yields the expected business process.

VI. Related Work

As stated before, dynamic adaptation has been a widely studied topic [14]. For instance, the MUSIC middleware [15] is defined to support component assembly self–adaptation. Event-based AOP (EAOP) is a framework that intends to express aspects in terms of events that arrive during execution [16]. They even detect sequences of events, and relate them using *event patterns* at run-time. However, these approaches for adaptation are only one way, and they never consider undoing their changes.

In [17], the authors present an aspect–oriented approach called WComp, a lightweight component–based middleware to design composite Web services. They propose an aspect-oriented approach called *Aspect of Assembly* (AA) to create a *composition for adaptation*. When a change in the context is detected, they create a simulation by applying all the AAs (implementing remaining adaptation rules) to the initial state and compare it to the actual state. Then they apply the differences by using pure elementary modifications (add, remove, link, unlink). Our approach is semantically different as they focus on one kind of event (*i.e.*, service apparition or vanishing) where we use the definition of CE to drive the (un)adaptation process.

In [18], the authors propose to automate the handling of model inconsistencies through the discovery of repair plans, implemented as action sequences. They demonstrate that action–based approaches support an *efficient* implementation of model manipulation. FScript [19] uses actions to support automated rollback (*i.e.*, only for reconfiguration failure). In [13], the authors propose an action–based approach to support the unweaving of model aspects. The underlying principles are close to the ones used in this proposal, *i.e.*, the execution of inverted action sequences and the replay of remaining adaptations (in this case, aspect application). However, our approach is different as we aim to automatically support the undoing of adaptations (without human intervention) where aspect model unweaving is a human–driven process. Moreover, the reification of relations between CEs helps us to smartly prune the encountered events and then implement an accurate replay.

In [20], the authors present their approach for creating dynamic business processes using ECA (Event-Condition-Action) rules. They decompose the original business process structure in a set of rules. This rules are then used to create a *Control Flow Checking* table, where the flow of the process is defined. To adapt the process they create a new modified *Control Flow Checking* table, which they compare to the original. The differences between both tables are then used to create new rules that will allow the new modifications to be considered during the BP execution. To undo the adaptation, the new rules could just be removed, or restored to their previous state. However, even though the goal of adaptation is accomplished, the introduction and removal of rules require external interaction, (*i.e.* somehow the rules need to be created and fed into the system).

VII. Conclusions and Future Work

In this paper we have argued that BP unadaptation is an important issue that should not be obviated, since naive approaches could end up with corrupted processes. We presented our approach to solve this issue with a four-step procedure: recognize, rewind, prune and replay. The proposed approach is generic since the detection of the undo trigger event is defined in terms of state–of–the–art CEP engines, and uses boolean logic to associate an event to its non–event. Furthermore, our approach is also automated. Indeed the way BPs are unadapted is fully delegated to an automatic engine. Thanks to these two advantages, we set the user free from having to deal with the unadaptation logic. Moreover, our four–step approach warranties that the final process will be as expected: the approach considers not only the original adaptation, but also the subsequent adaptations that were related to it, and unadapts them as well, while leaving the unrelated adaptations untouched. The obtained

result is a cleanly unadapted process, *"as if the original adaptation had never happened"*. We validated our approach by formalizing each step of the unadaptation procedure and implemented the logic approach with PROLOG.

As for future work, we are currently enhancing the proposed approach to support user–driven customization of the automatically generated unadaptation trigger events: these generated events could indeed be optimized according to business knowledge. We are also working in developing an instance level process (un)adaptation method. Since each process instance has a context of its own, adapting or unadapting cannot always be performed. To correctly achieve (un)adaptation at the instance level, we need to consider also the step of the process that the instance is running and then adapt its process only when referring to future steps. Also, a particular attention must be paid when loops are involved in the instance, since in this case a previous step is also a future step, and these adaptations need to be managed carefully.

ACKNOWLEDGEMENTS

This work was supported by the French Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the *Contrat de Projets Etat Region* (CPER) 2007-2013.

REFERENCES

[1] S. S. u. Rahman, N. Aoumeur, and G. Saake, "An Adaptive ECA-centric Architecture for Agile Service-based Business Processes with Compliant Aspectual .NET Environment," in *iiWAS '08: Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*. ACM, 2008, pp. 240–247.

[2] M. Koning, C.-a. Sun, M. Sinnema, and P. Avgeriou, "VxBPEL: Supporting variability for Web services in BPEL," *Inf. Softw. Technol.*, vol. 51, no. 2, pp. 258–269, 2009.

[3] A. Charfi, T. Dinkelaker, and M. Mezini, "A Plug-in Architecture for Self-Adaptive Web Service Compositions," in *ICWS '09: Proceedings of the 2009 IEEE International Conference on Web Services*. IEEE Computer Society, 2009, pp. 35–42.

[4] G. Hermosillo, L. Seinturier, and L. Duchien, "Creating Context-Adaptive Business Processes," in *ICSOC*, ser. Lecture Notes in Computer Science, P. P. Maglio, M. Weske, J. Yang, and M. Fantinato, Eds., vol. 6470, 2010, pp. 228–242.

[5] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[6] G. Hermosillo, L. Seinturier, and L. Duchien, "Using Complex Event Processing for Dynamic Business Process Adaptation," in *IEEE SCC*. IEEE Computer Society, 2010, pp. 466–473.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[8] G. Sharon and O. Etzion, "Event-processing Network Model and Implementation," *IBM Syst. J.*, vol. 47, pp. 321–334, April 2008. [Online]. Available: http://dx.doi.org/10.1147/sj.472.0321

[9] S. Mosser, "Behavioral Compositions in Service-Oriented Architecture," Ph.D. dissertation, University of Nice, Sophia–Antipolis, France, Oct. 2010. [Online]. Available: http://nyx.unice.fr/publis/mosser:2010.pdf

[10] OASIS, "Web Services Business Process Execution Language Version 2.0," OASIS, Tech. Rep., 2007. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.pdf

[11] X. Blanc, I. Mounier, A. Mougenot, and T. Mens, "Detecting Model Inconsistency through Operation-Based Model Construction," in *ICSE*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds. ACM, 2008, pp. 511–520.

[12] R. A. O'Keefe, *The craft of Prolog*. Cambridge, MA, USA: MIT Press, 1990.

[13] J. Klein, J. Kienzle, B. Morin, and J.-M. Jézéquel, "Aspect Model Unweaving," in *MoDELS*, ser. Lecture Notes in Computer Science, A. Schürr and B. Selic, Eds., vol. 5795. Springer, 2009, pp. 514–530.

[14] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds., *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science, vol. 5525. Springer, 2009.

[15] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. O. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz, "MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments," *Software Engineering for Self-Adaptive Systems*, vol. 5525, pp. 164–182, 2009.

[16] R. Douence, O. Motelet, and M. Südholt, "A Formal Definition of Crosscuts," in *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, ser. REFLECTION '01. Springer-Verlag, 2001, pp. 170–186.

[17] J.-Y. Tigli, S. Lavirotte, G. Rey, V. Hourdin, D. Cheung-Foo-Wo, E. Callegari, and M. Riveill, "WComp Middleware for Ubiquitous Computing: Aspects and Composite Event-based Web Services," *Annals of Telecommunications*, vol. 64, pp. 197–214, 2009.

[18] M. A. A. da Silva, A. Mougenot, X. Blanc, and R. Bendraou, "Towards Automated Inconsistency Handling in Design Models," in *CAiSE*, ser. Lecture Notes in Computer Science, B. Pernici, Ed., vol. 6051. Springer, 2010, pp. 348–362.

[19] M. Léger, T. Ledoux, and T. Coupaye, "Reliable Dynamic Reconfigurations in a Reflective Component Model," in *CBSE*, ser. Lecture Notes in Computer Science, L. Grunske, R. Reussner, and F. Plasil, Eds., vol. 6092. Springer, 2010, pp. 74–92.

[20] J. F. M. Bernal, P. Falcarin, M. Morisio, and J. Dai, "Dynamic Context-aware Business Process: a Rule-based Approach Supported by Pattern Identification," in *SAC*, S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, and C.-C. Hung, Eds. ACM, 2010, pp. 470–474.